FINAL COPY

Determining Fault Insertion Rates For Evolving Software Systems

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@jpl.nasa.gov

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson @cs.uidaho.edu

ABSTRACT

In developing a software system, we would like to be able to estimate the way in which the fault content changes during its development, as well as determining the locations having the highest concentration of faults. In the phases prior to test, however, there may be very little direct information regarding the number and location of faults. This lack of direct information requires the development of a fault surrogate from which the number of faults and their location can be estimated. We develop a fault surrogate based on changes in relative complexity, a synthetic measure which has been successfully used as a fault surrogate in previous work. We show that changes in the relative complexity can be used to estimate the rates at which faults are inserted into a system between successive revisions. These rates can be used to continuously monitor the total number of faults inserted into a system, the residual fault content, and identify those portions of a system requiring the application of additional fault detection and removal resources.

1. Introduction

Over a number of years of study, we can now establish a distinct relationship between software faults and certain aspects of software complexity. When a software system consisting of many distinct software modules is built for the first time, we have little or no direct information as to the location of faults in the code. Some modules will have far more faults in them then do others. However, we now know that the number of faults in a module is highly correlated with certain software attributes that may be measured. We can measure these attributes of the software and have some reasonable notion as to the degree to which the modules are fault prone [10, 14].

In the absence of information as to the specific location of software faults, we have successfully used a derived metric, the relative complexity measure, as a fault surrogate. That is, if the relative complexity value of a module is large, then it will likely have a large number of faults. If, on the other hand, the relative complexity of a module is small, then it will tend to have fewer faults. As the software system evolves through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code, however, is a fault prone process just as was the initial code generation. Faults may well be inserted during this evolutionary process.

Code does not always change just to fix faults that have been isolated in it. Some changes to code during its evolution represent enhancements, design modifications or changes in the code in response to continually evolving requirements. These incremental code enhancements may also result in the insertion of still more faults. Thus, as a system progresses through a series of builds, the relative complexity fault surrogate of each program module that has been altered must also change. We will see that the rate of change in relative complexity will serve as a good index of the rate of fault insertion.

Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise. We would be very mistaken, then, to assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault insertion, which in most cases is probably true [16]. The rate of fault removal is relatively easy to measure. The rate of fault insertion is much more tenuous. This fault insertion process is directly related to two measures that we can take on code as it evolves, code delta and code churn.

In this investigation we establish a methodology whereby code can be measured from one build to the next, a measurement baseline. We use this baseline to develop an assessment of the rate of change to a system as measured by our relative complexity fault surrogate. From this change process we are able to derive a direct measure of the rate of fault insertion based on changes in the software from one build to the next. Finally we examine data from an actual system on which faults may be traced to specific build increments to assess the predicted rate of fault insertion with the actual.

A major objective of this study is to identify a complete software system on which every version of every module has been archived together with the faults that have been recorded against the system as it evolved. For our purposes, the Cassini Orbiter Command and Data Subsystem (CDS) at JPL met our objectives. On the first build of this system there were approximately 96K source lines of code in approximately 750 program modules. On the last build there were approximately 110K lines of source code in approximately 800 program modules. As the system progressed from the first to the last build there were a total of 45,200 different versions of these modules. Each module progressed through an average of about 60 evolutionary steps or versions. For the purposes of this study, the Ada program module is a procedure or function; it is the smallest unit of the Ada language structure that may be measured. A number of modules present in the first build of the system were removed on subsequent builds. Similarly, a number of modules were added.

The Cassini CDS is quite typical of the amount of change activity that will occur in the development of a system on the order of 100 KLOC. It is a non-trivial measurement problem to track the system as it evolves. Again, there are two different sets of measurement activities that must occur at once. We are interested the changes in the source code and we are interested in the fault reports that are being filed against each module.

2. A Measurement Baseline

Measuring an evolving software system is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. This problem is very similar to that encountered by the surveying profession. To establish the topological characteristics of the land, we will have to seek out a benchmark. This benchmark represents an arbitrary point somewhere on the subject property. The distance and the elevation of every other point on the property may then be established in relation to the measurement baseline. Interestingly enough, we can pick any point on the property, establish a new baseline, and get exactly the same topology for the property. The property does not change. Only our perspective changes.

When measuring software evolution, we also need to establish a measurement baseline [18, 15]. We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property that, when another point is chosen, the exact same picture of software evolution emerges, only the perspective changes. The individual points involved in measuring software evolution are individual builds of the system.

In our land surveying analogy, however, there are only two attributes that we are concerned with, the height of a point relative to the baseline and the distance of that point from the baseline. Both attributes use the same measurement scale. Software attributes are very different. The raw measurements taken on the various attributes are all on different scales. The comparison of different modules within a software system by using raw measurement data is complicated by this fact. Take for example the data in Table 1. This table provides the values for two metrics; lines of code, *LOC*, and cyclomatic complexity, *V(g)*. These measurements are taken for two different builds of the system. Based on these two metrics, it is difficult to assert that Module A is more complex than Module B on Build 1. Certainly, *LOC* is less than that for module B, but *V(g)* is greater. Now consider the same two modules for build 2. Has the system, as represented by these two modules, become more complex or less complex between these two builds? The total number of lines of code has decreased by ten, but cyclomatic complexity has increased by two. Again, it is difficult to assert that there has been an increase or decrease in overall complexity. In order to make such comparisons it will be necessary to standardize the data.

	Build 1		**Build 2**	
Module	A	B	A	B
LOC	200	250	210	230
V(g)	20	15	19	18

Table 1. A Measurement Example

Standardizing metrics puts all of the metrics on the same relative scale, with a mean of zero and a standard deviation of one. However the standardization masks the change that has occurred between builds. In order to place all the metrics on the same relative scale and to keep from losing the effect of changes between builds, all build data is standardized using the means and standard deviations for the metrics obtained from the baseline system. This preserves trends in the data and lets measurements from different builds be compared.

Table 2 shows how a baseline may be established and used to compare software in different builds. In this table, the lines of code metrics for Modules A and B have been copied from the corresponding row of Table 1 to Table 2. We can see from these tables that Module A has increased 10 lines of code from Build 1 to Build 2. We can also see that Module B has decreased by 20 lines between these two

builds. What is not apparent from this table is the relative size of Modules A and B to other modules in the same build. To make this difference visible each of the *LOC* values is normalized by subtracting the mean value, $\overline{LOC}$, for each build, and dividing by the standard deviation of *LOC* for that build, s_{LOC}. This will yield the row labeled z_{LOC} in Table 2. With these normalized metric values, we can see that Module A has not changed in *LOC* relative to all other program modules. The same thing is true for Module B from Build 1 to Build 2. Module A is of average size on both Build 1 and Build 2. If, on the other hand, we normalize the Build 2 modules by the mean and standard deviation of Build 1, we obtain a new row for Table 2 labeled *Base* z_{LOC}. Build 2 may now be compared directly to Build 1. We can see that Module 2 is 0.4 standard deviations greater than it was on Build 1. Further, while Module B was fully two standard deviations above the mean *LOC* for Build 1, on Build 2 it has diminished to 1.2 standard deviations above the mean.

	Build 1		**Build 2**	
Module	A	B	A	B
LOC	200	250	210	230
z_{LOC}	0.0	2.0	0.0	2.0
Base z_{LOC}	0.0	2.0	0.4	1.2
$\overline{LOC}$	200		210	
s_{LOC}	25		15	

Table 2. A Baseline Example

For each raw metric in the baseline build, we may compute a mean and a standard deviation. Denote the vector of mean values for the baseline build as $\bar{\mathbf{x}}^B$ and the vector of standard deviations as $\mathbf{s}^B$. The standardized baseline metric values for any module j in an arbitrary build i may be derived from raw metric values as

$$z_j^{B,i} = \frac{w_j^{B,i} - \bar{x}_j^B}{s_j^B}.$$

Standardizing the raw metrics makes them more tractable. Among other things, it permits the comparison of metric values from one build to the next. However, standardization does not solve the main problem. There are too many metrics collected on each module over many builds. We have successfully used principal components analysis for reducing the dimensionality of the problem [15, 7]. This technique reduces a set of highly correlated metrics to a much smaller set of uncorrelated or orthogonal measures. One of the products of this technique is an orthogonal transformation matrix $\mathbf{T}$ that sends the standardized scores (the matrix $\mathbf{z}$) onto a reduced set of domain scores thusly, $\mathbf{d} = \mathbf{zT}$.

In the same manner as the baseline means and standard deviations are used to transform the raw metric of any build relative to a baseline build, the transformation matrix $\mathbf{T}^B$ derived from the baseline build is used in subsequent builds to transform standardized metric values obtained from that build to the reduced set of domain metrics as follows: $\mathbf{d}^{B,i} = \mathbf{z}^{B,i}\mathbf{T}^B$, where $\mathbf{z}^{B,i}$ are the standardized metric values from build i baselined on build B.

Another artifact of the principal components analysis is the set of eigenvalues that are generated for each of the new principal components. Associated with each of the new measurement domains is an eigenvalue, λ. These eigenvalues are large or small varying directly with the proportion of variance explained by each principal component. We have used these eigenvalues to create a new metric called relative complexity, ρ, that is the weighted sum of the domain metrics to wit: $\rho_i = 50 + 10\sum_{j=1}^{m}\lambda_j d_j$, where m is the dimensionality of the reduced metric set [10].

As was the case for the standardized metrics and the domain metrics, relative complexity may be baselined as well using the eigenvalues and the baselined domain values:

$$\rho_i^B = \sum_{j=1}^{m}\lambda_j^B d_j^B$$

If the raw metrics that are used to construct the relative complexity metric are carefully chosen for their relationship to software faults then the relative complexity metric will vary in exactly the same manner as the faults [13]. The relative complexity metric in this context is a fault surrogate. Whereas we cannot measure the faults in a program directly we can measure the relative complexity of the program modules that contain the faults. Those modules having a large relative complexity value will be found to be those with the largest number of faults [12].

3. Software Evolution

As program modules change from one build to another, the attributes of the changed program modules change. This means that there are measurable changes in modules from one build to the next. Each build is numerically and measurably different from its predecessor with respect to a particular set of metrics. Thus, the system must be re-measured whenever changes are made to it.

In order to describe the complexity of a system at each build, it will be necessary to know which version of each of the modules was a constituent in the program that failed. Consider a software system composed of n modules as follows: $m_1, m_2, m_3, \cdots, m_n$. Now, let m_j^i represent the i^{th} version of the j^{th} module. With this nomenclature, the first build of the system would be described by the set of modules: $< m_1^1, m_2^1, m_3^1, \cdots, m_n^1 >$. We can represent this configuration in a nomenclature that will permit us to de-

scribe the measurement process more precisely by recording the superscripts as vector elements in the following manner: $\mathbf{v}^i = <v_1^i, v_2^i, v_3^i, \cdots v_n^i>$. Thus, v_i^n in $\mathbf{v}^n$ represents the version number of the i^{th} module in the n^{th} build of the system. The cardinality of the set of elements in the vector $\mathbf{v}^n$ is determined by the number of program modules in the n^{th} build.

A natural way to capture the intermediate versions of the software is to have the system development occur under a configuration management system. All versions of all modules can be reconstructed from the time the program was placed under configuration control. That is, the precise nature of $\mathbf{v}^n$ can be determined from the configuration management system. A natural way to capture the intermediate measurements for each build would be to incorporate the measurement tools within the configuration management system. Just as code changes are maintained for each program module, so should code attribute changes be kept by the configuration management system. With these data, we will be able to assess the precise effect of the change from the build represented by $\mathbf{v}^i$ to $\mathbf{v}^{i+1}$ or even $\mathbf{v}^i$ to $\mathbf{v}^{i+k}$ or $\mathbf{v}^{i-k}$.

The change in the relative complexity in a single module between two builds may be measured in one of two distinct ways. First, we may simply compute the simple difference in the module relative complexity between build i and build j. We will call this value the code delta for the module m_a, or $\delta_a^{i,j} = \rho_a^{B,j} - \rho_a^{B,i}$. The absolute value of the code delta is a measure of code churn. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault insertion, removing a lot of code is probably as catastrophic as adding a bunch. The new measure of code churn, χ, for module m_a is simply $\chi_a^{i,j} = \left|\delta_a^{i,j}\right| = \left|\rho_a^{B,j} - \rho_a^{B,i}\right|$. The total net change of the system is the sum of the code delta's for a system between two builds i and j is given by $\Delta^{i,j} = \sum_{c \in \mathbf{v}^i} \delta_c^{i,j}$.

With a suitable baseline in place, and the module sets defined above, software evolution can be measured across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the change in system complexity as measured by a selected metric, code delta, or we can measure the total amount of change the system has undergone between builds, code churn.

A limitation of measuring code deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the code delta for the system will be close to zero. The overall complexity of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely affected by replacing the old modules with new ones. What we need is a measure to accompany code delta that indicates how much change has occurred. Code churn is a measurement, calculated in a similar manner to code delta, that provides this information. The net code churn of the same system over the same builds is

$$\nabla^{i,j} = \sum_{c \in \mathbf{v}^i} \chi_c^{i,j}.$$

When several modules are replaced between builds by modules of roughly the same complexity, code delta will be approximately zero but code churn will be equal to the sum of the value of ρ for all of the modules, both inserted and deleted. Both the code delta and code churn for a particular metric are needed to assess the evolution of a system.

4. Obtaining Average Build Values

Since relative complexity has clearly been established as a successful surrogate measure of software faults [11], it seems reasonable to use it as the measure against which we compare different builds. By definition, the average relative complexity, $\bar{\rho}$, of the baseline *system* will be

$$\bar{\rho}^B = \frac{1}{N^B} \sum_{i=1}^{N^B} \rho_i^B = 50,$$

where N^B is the cardinality of the set of modules on the baseline build B. Relative complexity for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The relative complexities are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average relative complexity for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The average relative complexity for subsequent builds is given by $\bar{\rho}^k = \frac{1}{N^k} \sum_{i=1}^{N^k} \rho_i^{B,k}$, where N^k is the cardinality of the set of program modules in the k^{th} build and $\rho_i^{B,k}$ is the baselined relative complexity for the i^{th} module of that set.

As code is modified over time, faults will be found and fixed. However, new faults will be introduced into the code as a result of the change. In fact, this fault insertion process is directly proportional to change in the program modules from one version to the next. Complexity will tend to increase as changes are made; only rarely will it decrease.

5. Definition Of A Fault

Unfortunately there is no particular definition of just precisely what a software fault is. In the face of this diffi-

culty it is rather hard to develop meaningful associative models between faults and metrics. Since structural measurements are made at the module level, we would like information about faults at the same granularity.

Following the second definition of fault in [3, 4], we consider a fault to be a **structural imperfection** in a software system that **may** lead to the system's eventually failing. It is a **physical characteristic** of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. Faults are inserted into a system by people making errors in their tasks - these may be errors of commission or errors of omission.

In order to count faults, we needed to develop a method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our structural measurements. Faults may be localized to a single module – for instance, the order in which two blocks are required to execute may be reversed. Faults may also span multiple modules - for instance, each module containing a faulty include file would have those faults. Another example would be a faulty global data definition - each module referencing that global data item would contain the fault associated with that data item. Both types of faults must be taken into account.

For the Cassini CDS flight software, the failure information was obtained from the JPL institutional problem reporting system. Failures were recorded in this system starting at subsystem-level integration, and continuing through spacecraft integration and test. Failure reports typically contain descriptions of the failure at varying levels of detail, as well as descriptions of what was done to correct the fault(s) that caused the failure. Detailed information regarding the underlying faults (e.g., where were the code changes made in each affected module) is generally unavailable from the problem reporting system.

The Cassini CDS flight software development effort used the Software Configuration Control System (SCCS) to control changes to the software during its development. When a module was created, or changed in response to a failure report or engineering change request, the file containing the module was checked into SCCS as a new increment. This allowed us to track changes to the system at the module level as it evolved. For approximately 10% of the failure reports, we were able to identify the source file increment in which the fault(s) associated with a particular failure report were repaired. This information was available either in the comments inserted by the developer into the SCCS file during check-in, or in the comments at the beginning of a module that track its development history.

Using the information described above, we performed the following steps to identify faults. First, for each problem report, we searched all of the SCCS files to identify all modules and the increment(s) of each module for which the software was changed in response to the problem report. Second, for each increment of each module identified in the previous step, we assumed as a starting point that all differences between the increment in which repairs are implemented and the previous increment are due solely to fault repair. This is not necessarily a valid assumption - developers may be making functional enhancements to the system in the same increment that fault repairs are being made. Careful analysis of failure reports for which there was sufficiently detailed descriptive information served to separate areas of fault repair from other changes. However, the level of detail required to perform this analysis was not consistently available. Third, we used a differential comparator (e.g., Unix `diff`) to show the differences between the increment(s) in which the fault(s) were repaired, and the immediately preceding increment(s). The results indicated the areas to be searched for faults.

After completing the last step, we still had to identify and count the faults - the results of the differential comparison cannot simply be counted up to give a total number of faults. In order to do this, we developed a fault taxonomy [19]. This taxonomy differs from others in that it does not seek to identify the root cause of the fault. Rather, it is based on the types of changes made to the software to repair the faults associated with failure reports - in other words, it constitutes an operational definition of a fault. Although identifying the root causes of faults is important in improving the development process [1, 5], it is first necessary to identify the faults. Our taxonomy allowed us to identify faults in a consistent manner at the module level.

6. The Relationship Between Faults And Code Changes

To determine the relationship between structural change and the number of faults inserted, two measurement activities were performed. First, all of the versions of all of the source code modules were measured. From these measurements, code churn and code deltas were obtained for every version of every module. The failure reports were sampled to lead to specific faults in the code. These faults were classified according to our taxonomy manually on a case by case basis. We then developed regression models relating code measures to code faults.

The Ada source code for all versions of each of these modules was systematically reconstructed from the SCCS code deltas. Each of these module versions was then measured by the UX-Metric analysis tool for Ada [20]. Only a subset of metrics provided by this tool actually provide distinct sources of variation [6]. The specific metrics used are shown in Table 3.

To establish a baseline system, all of the metric data for the module versions that were members of the first build of CDS were analyzed by our PCA-RCM tool. This tool is designed to compute relative complexity values either from

a baseline system or from a system being compared to the baseline system. In that the first build of the Cassini CDS system was selected to be the baseline system, the PCA-RCM tool performed a principal components analysis on these data with an orthogonal varimax rotation. The objective of this phase of the analysis is to use the principal components technique to reduce the dimensionality of the metric set. As may been seen in Table 4, there are four principal components for the 18 metrics shown in Table 3. For convenience, we have named these principal components **Size**, **Structure**, **Style** and **Nesting**. From the last row in Table 4 we see that the new reduced set of orthogonal components of the original 18 metrics account for about 85% of the variation in the original metric set.

Metrics	Definition
η_1	Count of unique operators [2]
η_2	Count of unique operands
N_1	Count of total operators
N_2	Count of total operands
P/R	Purity ratio: ratio of Halstead's $\hat{N}$ to total program vocabulary
V(g)	McCabe's cyclomatic complexity
Depth	Maximum nesting level of program blocks
AveDepth	Average nesting level of program blocks
LOC	Number of lines of code
Blk	Number of blank lines
Cmt	Count of comments
CmtWds	Total words used in all comments
Stmts	Count of executable statements
LSS	Number of logical source statements
PSS	Number of physical source statements
NonEx	Number of non-executable statements
AveSpan	Average number of lines of code between references to each variable
Vl	Average variable name length

Table 3. Software Metric Definitions

As is typical in the principal components analysis of metric data, the **Size** domain dominates the analysis. It alone accounts for approximately 38% of the total variation in the original metric set. Not surprisingly, this domain contains the metrics of total statement count (*Stmts*), logical source statements (LSS), the Halstead lexical metric primitives of operator and operand count, but it also contains cyclomatic complexity (*V(g)*). In that we regularly find cyclomatic complexity in this domain we are forced to conclude that it is only a simple measure of size in the same manner as statement count. The **Structure** domain contains those metrics relating to the physical structure of the program such as non-executable statements (*NonEx*) and the program block count (*Blk*). The **Style** domain contains measures of attribute that are directly under a programmer's control such as variable length (*Vl*) and purity ratio (*P/R*). The **Nesting** domain consist of the single metric that is a measure of the average depth of nesting of program modules (*AveDepth*).

In order to transform the raw metrics for each module version into their corresponding relative complexity values, the means and the standard deviations must be computed. These are shown in Table 5. These values are used to transform all raw metric values for all versions of all modules to their baselined z score values. The last four columns of Table 5, D1, D2, D3, and D4, contain the actual transformation matrix that maps the metric z score values onto their orthogonal equivalents to obtain the orthogonal domain metric values used in the computation of relative complexity. The eigenvalues for the four domains are given in the last row of this table.

Metric	Size	Structure	Style	Nesting
Stmts	0.968	0.022	-0.079	0.021
LSS	0.961	0.025	-0.080	0.004
N_2	**0.926**	0.016	0.086	0.086
N_1	**0.934**	0.016	0.074	0.077
η_2	**0.884**	0.012	-0.244	0.043
AveSpan	**0.852**	0.032	0.031	-0.082
V(g)	**0.843**	0.032	-0.094	-0.114
η_1	**0.635**	-0.055	-0.522	-0.136
Depth	**0.617**	-0.022	-0.337	-0.379
LOC	-0.027	**0.979**	0.136	0.015
Cmt	-0.046	**0.970**	0.108	0.004
PSS	-0.043	**0.961**	0.149	0.019
CmtWds	0.033	**0.931**	0.058	-0.010
NonEx	-0.053	**0.928**	0.076	-0.009
Blk	0.263	**0.898**	0.048	0.005
P/R	-0.148	-0.198	**-0.878**	0.052
Vl	0.372	-0.232	**-0.752**	0.010
AveDepth	-0.000	-0.009	0.041	**-0.938**
% Variance	37.956	30.315	10.454	6.009

Table 4. Software Metrics Principal Components

Table 5 contains all of the essential information needed to obtain baselined relative complexity values for any version of any module relative to the baseline build. Once the baselined relative complexity data have been assembled for all versions of all modules, it is possible to examine some trends that have occurred during the evolution of the system. For example, Figure 1 shows the relative complexity of the evolving CDS system across one of its five major builds. To compute these values, every development increment within that build was identified. Then, for each increment, the baselined relative complexity values of the modules in that

increment were computed. The next four builds, not shown here, have evolutionary patterns similar to that shown in Figure 1. The average relative complexity of most systems seems to be a monotonically increasing function.

Not all program modules received the same degree of modification as the system evolved. Figure 2 shows the code churn and code delta values for a module that changed very little over its history. There were only four relatively minor changes to this module. A more typical change history is shown for another module in Figure 3. The total code churn for this module is approximately 38. Note that the net code delta for this module is close to zero - the relative complexity of the module at the last version is very close to its original relative complexity. This figure clearly illustrates the conceptual differences between the two measures of code churn and code delta. Code churn is a monotonically increasing value over sequential versions.

Metric	$\bar{x}^B$	$\ddot{a}^B$	D1	D2	D3	D4
Stmts	11.37	7.79	0.10	-0.02	0.26	0.05
LSS	25.18	27.08	0.13	0.00	0.04	-0.09
N_2	79.59	129.08	0.13	0.02	-0.17	-0.08
N_1	68.24	115.72	0.13	0.02	-0.17	-0.09
η_2	1.32	0.54	0.00	-0.07	0.54	-0.16
Ave-Span	4.77	6.19	0.12	0.01	-0.03	0.07
V(g)	1.48	1.58	0.10	-0.01	0.17	0.30
η_1	0.00	0.05	0.01	0.00	0.06	0.88
Depth	162.05	515.83	-0.01	0.17	0.07	-0.02
LOC	19.05	30.14	0.03	0.16	0.07	-0.02
Cmt	34.19	124.24	-0.01	0.17	0.09	-0.01
PSS	139.27	452.48	0.00	0.16	0.10	0.00
Cmt Wds	16.61	20.44	0.14	0.01	-0.07	-0.05
Non Ex	17.52	23.50	0.14	0.01	-0.07	-0.04
Blk	108.80	372.11	-0.01	0.17	0.06	-0.02
P/R	7.36	22.84	-0.01	0.16	0.10	0.00
Vl	5.75	8.26	0.12	0.02	-0.11	0.06
Ave Depth	9.00	4.40	0.07	-0.06	0.40	-0.11
Eigen-values			6.832	5.457	1.882	1.082

Table 5. Baseline Transformation Data

Figure 4 shows a module at the extreme end of change history. This module has a total code churn value of close to 140. Also, its final code delta value is about 30 indicating that its net relative complexity has also increased substantially as it evolved. Among the three modules whose change history is illustrated by Figures 2, 3, and 4, the latter module is the one on which we focus the most attention. It is the one most likely to have had large numbers of faults inserted into it throughout its dramatic life.

Note in Figure 1 that not all increments within a build represent the same increase in relative complexity. Nearly one third of the total change in this version takes place within the first 10% of the development increments. From our understanding of the relationship between relative complexity as a fault surrogate and inserted faults, we would expect a large number of faults to have been inserted during the first 30 increments because of the amount of change that occurred. It is also interesting to note that the final relative complexity of this version is rather close to the initial relative complexity, although it is clear from the measured code churn that a significant amount of change has occurred.

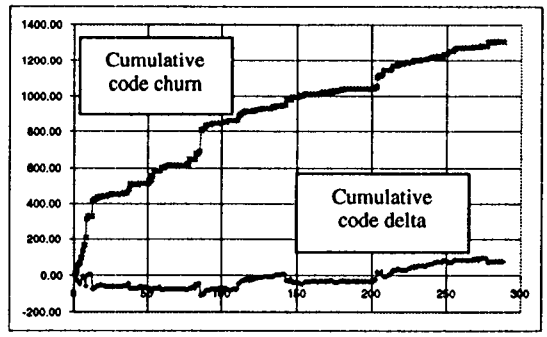

Figure 1. Change in Relative Complexity for One Version of CDS Flight Software

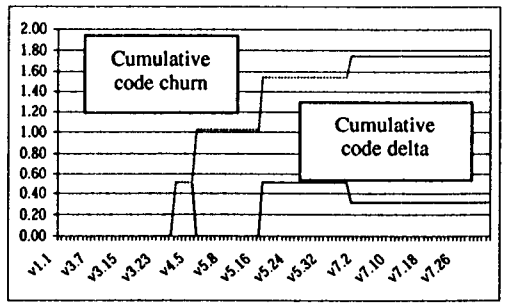

Figure 2. Change History for Stable Module

In relating the number of faults inserted in an increment to measures of a module's structural change, we had only a small number of observations with which to work. There were three difficulties that had to be dealt with. First, although over 600 failure reports were written against the Cassini CDS flight software during developmental testing and system integration, for only about 10% of the failure reports were we able to identify the module(s) that had been changed, and in which increment those changes were made. Second, once a fault had been identified, it was necessary to trace it back to the increment in which it first occurred.

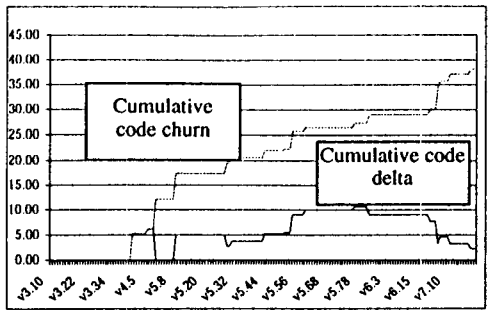

Figure 3. Typical Module Change History

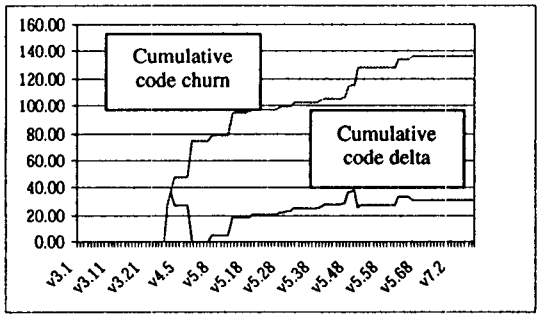

Figure 4. Change History for Frequently Changed Module

For some source files, there were over 100 increments that had to be manually searched. Since the SCCS files for each delivered version were available, most faults could be traced to their point of origin. The principal difficulty was the volume of material being examined – this was one of the factors restricting the number of observations that could be obtained. Third, there were instances in which the UX-Metric analyzer would not provide the structural measurements of a module. The result was that of the over 100 faults initially identified, there were only 35 observations in which a fault could be associated with a particular increment of a module, and with that increment's measures of code delta and code churn.

For each of the 35 instances of viable fault data, there were three data points. First, we had a count of the number of faults inserted into a particular increment i of a module m_a. Second, we had code delta values, $\delta_a^{i-1,i}$, for each of these modules. Finally, we had code churn values, $\chi_a^{i-1,i}$, derived from the code deltas.

Linear regression models were computed for code churn and code deltas with code faults as the dependent variable in both cases. Both models were built without constant terms; we assumed that if no changes were made to a module, then no new faults could be introduced. The results of the regression between faults and code deltas were not surprising. The squared multiple R for this model was 0.001, about as close to zero as you can get. This result is directly attributable to the non-linearity of the data. Change may increase the complexity of a module, or it may decrease a module's complexity. Faults, on the other hand are not primarily related to the direction of the change but to its intensity. Removing code from a module is just as likely to introduce faults as adding code to it.

The relationship between code churn and faults is dramatically different. The regression ANOVA for this model are shown in Table 6. Whereas code deltas do not show a linear relationship with faults, code churn certainly does. The regression model is given in Table 7; the regression statistics are reported in Table 8. Of particular interest is the Squared Multiple R term. This has a value of 0.653. This means, roughly, that the regression model will account for more that 65% of the variation in the faults of the observed modules based on the values of code churn.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	331.879	1	331.879	62.996	0.000
Residual	179.121	34	10.673	5.268	

Table 6. Regression Analysis of Variance

Effect	Coefficient	Std Err	t	P(2-Tail)
Churn	0.576	0.073	7.937	0.000

Table 7. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	0.806	0.649	2.296

Table 8. Regression Statistics

It may be the case that both the amount of change and the direction in which the change occurred affect the number of faults inserted. The linear regression through the origin, shown in Tables 9, 10, and 11 is based on this idea. This model, incorporating code delta and code churn, performs substantially better than the model incorporating only code churn, as measured by Squared Multiple R and Mean Sum of Squares.

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	367.247	2	183.623	42.153	0.000
Residual	143.753	33	4.356		

Table 9. Regression Analysis of Variance

We wanted to see if a non-linear relationship between measurements of a system's evolution and the number of faults inserted might be more appropriate. We developed two non-linear regression models describing the number of faults inserted as functions of code churn and delta:

$$d^{j,j+1} = b_0 * \left(\nabla^{j,j+1}\right)^{b_1}, \quad d^{j,j+1} = b_0 * \left(\nabla^{j,j+1}\right)^{b_1} * \left(b_2\right)^{\Delta^{j,j+1}}$$

where $d^{j,j+1}$ represents the number of faults inserted between revisions j and j+1, $\nabla^{j,j+1}$ represents the amount of code churn between builds j and j+1, and $\Delta^{j,j+1}$ represents the code delta between revisions j and j+1.

Effect	Coefficient	Std Err	t	P(2-Tail)
Churn	0.647	0.071	9.172	0.000
Delta	0.201	0.071	2.849	0.002

Table 10. Regression Model

N	Multiple R	Squared multiple R	Standard error of estimate
35	.848	.719	2.087

Table 11. Regression Statistics

Because Multiple Squared R values for linear regressions through the origin cannot be compared with Multiple Squared R values for the non-linear regressions, we used Predicted Residual Sum of Squares (PRESS) to compare the models, as was done in [9]. Models with lower PRESS scores produce better predictions. To compute PRESS, delete in turn each observation in a set of observations and fit a model M to the remaining observations. The associated prediction at the value for the deleted observation is compared to the actual value of that observation.

Table 12 gives PRESS scores for the linear regressions through the origin and both non-linear regressions. The four observations for which the value of code churn was zero were excluded from the regressions, since the non-linear regressions could not produce estimates with these observations. The linear model incorporating only code churn has a substantially lower PRESS score than the corresponding non-linear model. Although the non-linear model using both code churn code and code delta has a lower PRESS score than the corresponding linear model, the difference is not large enough to indicate a clear preference between the two models. This indicates that the relationship between measures of system evolution and the number of faults inserted may be more appropriately expressed as a linear rather than a non-linear relationship.

Effect	Linear Models Excluding Observations of Churn = 0	Nonlinear Models Excluding Observations of Churn = 0
Churn only	186.090	205.718
Churn and Delta	159.875	157.831

Table 12. PRESS Scores for Linear and Nonlinear Regressions

Finally, we investigated whether the linear regression model which uses code churn alone is an adequate predictor at a particular significance level when compared to the model using both code churn and code delta. We used the R^2-adequate test [8, 17] to examine the linear regression models through the origin and determine whether the model that depends only on code churn is an adequate predictor. A subset of predictor variables is said to be R^2-adequate at significance level α if:

$R^2_{sub} > 1 - (1 - R^2_{full})(1 + d_{n,k})$, where

- R^2_{sub} is the R^2 value for the subset of predictors
- R^2_{full} is the R^2 value for the full set of predictors
- $d_{n,k} = (kF_{k,n-k-1})/n-k-1$, where
  - k = number of predictor variables in the model
  - n = number of observations
  - F = F statistic for significance α for n,k degrees of freedom.

Table 13 shows values of R^2, k, degrees of freedom, $F_{k,n-k-1}$, $d_{n,k}$, and R^2_{sub} for both linear models through the origin. The number of observations, n, is 35, and we specify α=.05.

Lin. Regressions Through Origin	R^2	DF	k	$F_{k,n-k-1}$ for significance α	d(n,k)	Threshold for significance α
Churn only	0.649	34	1	4.139	0.125	-----
Churn, Delta	0.719	33	2	3.295	0.206	0.661

Table 13 – Values of R^2, DOF, k, $F_{k,n-k-1}$, and $d_{n,k}$ for R^2-adequate Test

Table 13 shows that the value of Multiple Squared R for the regression using only code churn is 0.649. The 5% significance threshold for the code churn and code delta model is 0.661. This means that the regression model using only code churn is not R^2 adequate when compared to the model using both code churn and code delta. Although the amount of change occurring between subsequent revisions appears to be the primary factor determining the number of faults inserted, the direction of that change also appears to be a significant factor.

7. Summary

There is a distinct and a strong relationship between software faults and measurable software attributes. The most interesting result of this current endeavor is that we also found a strong association between the fault insertion process over the evolutionary history of a software system and the degree of change that is taking place in each of the program modules. We also found that the direction of the change was significant in determining the number of faults inserted. Some changes will have the potential of introducing very few faults while others may have a serious impact on the number of latent faults.

In order for the measurement process to be meaningful, the fault data must be very carefully collected. In this study, the data were extracted ex post facto as a very labor intensive effort. A well defined fault standard and fault taxonomy must be developed and maintained as part of the software development process. Further, all designers and coders should be thoroughly trained in its use. A viable standard is one that may be used to classify any fault unambiguously. A viable fault recording process is one in which any one person will classify a fault exactly the same as any other person.

Finally, the whole notion of measuring the fault insertion process is its ultimate value as a measure of software process. The techniques developed in this study can be implemented in a development organization to provide a consistent method of measuring fault content and structural evolution across multiple projects over time. The initial estimates of fault insertion rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault insertion rate, and to identify those development techniques that seem to provide the greatest reduction.

Acknowledgments

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology. Portions of the work were sponsored by the National Aeronautics and Space Administration's IV&V Facility and the U. S. Air Force Operational Test and Evaluation Center (AFOTEC).

References

[1] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurement", IEEE Transactions on Software Engineering, November, 1992, pp. 943-946.

[2] M. H. Halstead, *Elements of Software Science*. Elsevier, New York, 1977..

[3] "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[4] "IEEE Standard Dictionary of Measures to Produce Reliable Software", IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.

[5] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994

[6] T. M. Khoshgoftaar and J. C. Munson , "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* 8, 1990, pp. 253-261.

[7] T. M. Khoshgoftaar and J. C. Munson "A Measure of Software System Complexity and Its Relationship to Faults," In *Proceedings of the 1992 International Simulation Technology Conference*, The Society for Computer Simulation, San Diego, CA, 1992, pp. 267-272.

[8] S. G. MacDonell, M. J. Shepperd, P. J. Sallis, "Metrics for Database Systems: An Empirical Study", *Proceedings of the Fourth International Software Metrics Symposium*, November 5-7, 1997, Albuquerque, NM, pp. 99-107

[9] J. A. Morgan and G. J. Knafl, "Residual Fault Density Prediction using Regression Methods", *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, White Plains, NY, October 1996, pp. 87-92.

[10] J. C. Munson and T. M. Khoshgoftaar "Regression Modeling of Software Quality: An Empirical Investigation," *Journal of Information and Software Technology*, 32, 1990, pp. 105-114.

[11] J. C. Munson and T. M. Khoshgoftaar "The Relative Software Complexity Metric: A Validation Study," In *Proceedings of the Software Engineering 1990 Conference*, Cambridge University Press, Cambridge, UK, 1990, pp. 89-102.

[12] J. C. Munson and T. M. Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18, No. 5, 1992, pp. 423-433.

[13] J. C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering*, J. C. Baltzer AG, Amsterdam 1995.

[14] J. C. Munson, "Software Faults, Software Failures, and Software Reliability Modeling", *Information and Software Technology*, December, 1996.

[15] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium*, IEEE Computer Society Press, pp. 41-51.

[16] J. C. Munson and G. A. Hall, "Estimating Test Effectiveness with Dynamic Complexity Measurement," *Empirical Software Engineering Journal*. Feb. 1997.

[17] J. Neter, W. Wasserman, M. H. Kutner, Applied Linear Regression Models, Irwin: Homewood, IL, 1983

[18] A. P. Nikora, N. F. Schneidewind, J. C. Munson, "IV&V Issues in Achieving High Reliability and Safety in Critical Control System Software", *Proceedings of the International Society of Science and Applied Technology Conference*, March 10-12, 1997, Anaheim, CA, pp 25-30.

[19] A. P. Nikora, "Software System Defect Content Prediction From Development Process And Product Characteristics", Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[20] "User's Guide for UX-Metric 4.0 for Ada", SET Laboratories, Mulino, OR, © SET Laboratories, 1987-1993.